

# Exploring performance and power properties of modern multicore chips via simple machine models

G. Hager, J. Treibig, J. Habich, and G. Wellein  
 Erlangen Regional Computing Center (RRZE)  
 Martensstr. 1, 91058 Erlangen, Germany

August 13, 2012

## Abstract

Modern multicore chips show complex behavior with respect to performance and power. Starting with the Intel Sandy Bridge processor, it has become possible to directly measure the power dissipation of a CPU chip and correlate this data with the performance properties of the running code. We establish machine models that describe the interaction of parallel code with the hardware, going beyond a simple bottleneck analysis. Together with a phenomenological power model we are able to explain many peculiarities in the performance and power behavior of multicore processors, and derive guidelines for power-efficient execution of parallel programs.

## 1 Introduction and related work

The transition to multicore technology in mainstream scientific computing has led to a plethora of new performance effects and optimization approaches. Since Moore’s law is alive and well, we may expect growing core counts at least in the mid-term future, together with the ubiquitous memory bandwidth bottleneck. Hence, sensitivity (or lack thereof) to limited memory bandwidth is one of the essential traits that can be used to characterize the performance of scientific simulation codes. Any serious “white-box” performance modeling effort with a claim to explain some particular performance aspect on the core, chip, and node levels must be able to address the interaction of code with inherent architectural bottlenecks, of which bandwidth or, more generally, data transfer, is the most important one. The balance metric [1] and its refined successor, the roofline model [2], are very successful instruments for predicting the bandwidth-boundedness of computational loops. Beyond clear bottlenecks, successful performance modeling can become very complex; even if cycle-accurate simulators were available, understanding the cause of a particular performance issue may require a grasp of computer architecture that the average computational scientist, as a mere user of compute resources, does not have. Simple machine models that build on published, assessable information about processor architecture can thus be very helpful also for the non-expert. Whenever the model fails to describe some performance feature within some margin of accuracy, there is the opportunity to learn something new about the interaction of hardware and software. The Execution-Cache-Memory (ECM) model [3] is a starting point for refined performance modeling of streaming loop kernels that also allows a well-founded prediction of the saturation point in case of strong bandwidth limitation, which was previously addressed in a more phenomenological way [4]. Performance modeling is certainly not limited to the single node, and there are many examples of successful large-scale modeling efforts [5, 6, 7, 8]. However, the chip is where the “useful code” that solves an actual problem is executed, and this is where insight into performance issues starts. This paper is restricted to the multicore chip level.

Besides performance aspects, considerations on power dissipation of multicore chips have become more popular in recent years; after all, concerns about the power envelope of high-end processors have led to the introduction of multicore in the first place. It is highly conceivable that some future processor designs will not be faster

but “only” more power-efficient than their predecessors. This development has already started in the mobile and embedded market, but will soon hit scientific computing as well. The latest x86 architectures have many power efficiency features built in, such as the ability to go to “deep sleep” states, to quickly switch off parts of the logic that are currently unused, and to speed up or slow down the clock frequency together with the core voltage. A lot of research has been devoted to using these features creatively, e.g., with dynamic voltage/frequency scaling (DVFS) or dynamic concurrency throttling (DCT) [9, 4, 10], in order to either save energy or limit the power envelope of highly parallel systems.

The Intel Sandy Bridge processor exposes some of its power characteristics to the programmer by means of its “Running Average Power Limit” (RAPL) feature. Among other things, it enables a measurement of the power dissipation of the whole chip with millisecond resolution and decent accuracy [11, 12].

Taken together, the performance and power dissipation properties of modern multicore chips are an interesting field for modeling approaches. We try to explore both dimensions here by using simple machine models that can be employed by domain scientists but are still able to make predictions with sufficient accuracy to be truly useful. This work is organized as follows: After an introduction of the used hard- and software in Sect. 2, we turn to the ECM model in Sect. 3 and apply it to streaming kernels and to multicore-parallel execution. In Sect. 4 we first study the power dissipation characteristics of several benchmark codes. Simplifying from these results, we derive an elementary power model that makes predictions about the “energy to solution” metric and its behavior with clock frequency, serial code performance, and the number of used cores. We validate the model against measurements on the chosen benchmark codes. A lattice-Boltzmann flow solver is then used in Sect. 5 as a test case for the performance and power models. Sect. 6 summarizes the paper and gives an outlook to future research.

## 2 Test bed and tools

### 2.1 Hardware

Unless otherwise noted, we have used a dual-socket eight-core Intel Sandy Bridge EP platform for single-node measurements and validation (Xeon E5-2680, 2.7 GHz base clock speed, turbo mode up to 3.5 GHz). The Intel Sandy Bridge microarchitecture contains numerous enhancements in comparison to its predecessor Westmere. The following features are most important for our analysis [13]:

- The AVX (Advanced Vector Extensions) instruction set extension doubles the SIMD register width from 128 to 256 bits. At the same time, the load throughput of the L1 cache was doubled from 16 bytes to 32 bytes per cycle, so that a Sandy Bridge core can sustain one full-width AVX load and one half-width AVX store per cycle.
- The core can execute one ADD and one MULT instruction per cycle (pipelined). With double-precision AVX instructions, this leads to a peak performance of eight flops per cycle (sixteen at single precision). In general, the core has a maximum instruction throughput of six  $\mu$ ops per cycle.
- The L2 cache sustains refills and evicts to and from L1 at 256 bits per cycle (half-duplex). A full 64-byte cache line refill or evict thus takes two cycles. This is the same as on earlier Intel designs.
- The L3 cache is segmented, with one segment per core. All segments are connected by a ring bus. Each segment has the same bandwidth capabilities as the L2 cache, i.e., it can sustain 256 bits per cycle (half-duplex) for refills and evicts from L2. This means that the L3 cache is usually not a bandwidth bottleneck and streaming loop kernels show good scaling behavior when the data resides in L3.
- One Xeon E5-2680 socket of our test systems has four DDR3-1333 memory channels for a theoretical peak bandwidth of 42.7 GB/s. In practice, 36 GB/s can be achieved with the STREAM benchmark. This translates to 107 bits per core cycle (at base clock speed).
- Each of the two E5-2680 sockets forms its own ccNUMA locality domain. They are connected via a 8 GT/s QPI link. Since all our benchmarks and application codes use parallel first-touch page placement, there are no issues with ccNUMA contention and nonlocality effects.

## 2.2 Tools

We have used the Intel compiler Version 12.1 update 9 for compiling source codes. Hardware counter measurements were performed with the `likwid-perfctr` tool from the LIKWID tool suite [14, 15], which, in its latest development release, can access the power information (via the RAPL interface [11]) and the “uncore” events (i.e., L3 cache and memory/QuickPath interface) on Sandy Bridge processors.

The LIKWID suite also contains `likwid-bench` [16], a microbenchmarking framework that makes it easy to build and run assembly language loop kernels from scratch, without the uncertainties of compiler code generation. `likwid-bench` was used to validate the results for some of the streaming microbenchmarks in this work.

## 3 A refined machine model for streaming loop kernels on multicore

The majority of numerical codes in computational science are based on streaming loop kernels, which show good data parallelism and are largely compatible with the hardware prefetching capabilities of modern multicore processors. For large data sets, such kernels are often (but not always) bound by memory bandwidth, which leads to a peculiar scaling behavior across the cores of a multicore chip: Up to some critical number of cores  $t_s$  scalability is good, but for  $t > t_s$  performance saturates and is capped by some maximum level. Beyond the saturation point, the roofline model [2] can often be used to predict the performance, or at least its qualitative behavior with respect to problem parameters, but it does not encompass effects that occur between the cache levels. The “Execution-Cache-Memory” (ECM) model [3, 17] adds basic knowledge about the cache bandwidths and organization on the multicore chip to arrive at a more accurate description on the single-core level. Although the model can be used to predict the serial and parallel performance of codes on multicore processors, its main purpose is to develop a deeper understanding of the interaction of code with the hardware. This happens when the model *fails to coincide* with the measurement (see Sect. 3.1 below).

In the following we give a brief account of this model, show how it connects to the roofline model, apply it to parallel streaming kernels, and refine it to account for some unknown (or undisclosed) properties of the cache hierarchy. In Sect. 5 we apply the model to a lattice-Boltzmann flow solver.

### 3.1 The Execution-Cache-Memory (ECM) model: Single core

The main premise of the ECM model is that the runtime of a loop is composed of two contributions: (i) The time it takes to execute all instructions, with all operands of loads or stores coming from or going to the L1 data cache. We call this “core time.” (ii) The time it takes to transfer the required cache lines into and out of the L1 cache. We call this “data delays.” The model further assumes that hardware or software prefetching mechanisms are in place that hide all cache transfer latencies.

Since all data transfers between cache levels occur in packets of one cache line, the model always considers one cache line’s worth of work. For instance, if a double precision array must be read with unit stride for processing, the basic unit of work in the model is eight iterations at a cache line size of 64 bytes. The execution time for one unit of work is then composed of the in-core part  $T_{\text{core}}$  and the data delays  $T_{\text{data}}$ , with potential overlap between them.

The further away from the L1 a needed cache line resides, the larger  $T_{\text{data}}$ , since it takes time to transfer cache lines up to L1 or back. Note that, since we have assumed perfect prefetching, this is not a simple latency effect: It comes about because of limited bandwidth and several possibly non-overlapping contributions. For example, on a Sandy Bridge core, the transfer of a 64-byte cache line from L3 through L2 to L1 takes a maximum of four and a minimum of two cycles (32-byte wide buses between the cache levels), depending on whether the transfers can overlap. Furthermore, the L1 cache of Intel processors is “single-ported” in the sense that, in any clock cycle, it can either reload/evict cache lines from/to L2 or communicate with the registers, but not both at the same time.

The core time  $T_{\text{core}}$  is more complex to estimate. In the simplest case, execution is dominated by a clear bottleneck, such as load/store throughput or pipeline stalls. Some knowledge about the core microarchitecture like the kind and number of execution ports or the maximum instruction throughput is helpful for getting a first estimate. For example, in a code that is completely dominated by independent ADD instructions, the core time

is, to first order, determined by the ADD port throughput (one ADD instruction per cycle on modern Intel CPUs). In a complex loop body, however, it is often hard to find the critical execution path that determines the number of cycles. The Intel Architecture Code Analyzer (IACA) [18] is a tool that can derive more accurate predictions by taking dependencies into account. See, e.g., [17] for a detailed analysis of a complex loop body with IACA.

Putting together a prediction for the overall execution time requires making best- and worst-case assumptions about possible overlaps of the different contributions described above. If the measured performance data is far off those predictions, the model misses an important architectural or execution detail, and must be refined. A simple example is the write-allocate transfer on a store miss: A naive model for the execution of a store-dominated streaming kernel (like, e.g., array initialization  $A(:)=0$ ) with data in the L2 cache will predict a bandwidth level that is much higher than the measurement. Only when taking into account that every cache line must be transferred to L1 first will the prediction be correct.

On many of today's multicore chips a single core cannot saturate the memory interface, although a simple comparison of peak performance vs. memory bandwidth suggests otherwise: An Intel Sandy Bridge core, for example, has a (double precision) arithmetic peak performance of  $P_{\text{peak}} = 21.6 \text{ GF/s}$  at 2.7 GHz, and the maximum memory bandwidth of the chip is 36 GB/s (see Sect. 2.1). This leads to a machine balance of  $B_c = 1.7 \text{ B/F}$ . The code balance of the triad loop from the STREAM benchmarks ( $A(:)=s*B(:)+C(:)$ ) is  $B_m = 16 \text{ B/F}$  (including the write-allocate for  $A(:)$ ), so the roofline model predicts a strong limitation from memory bandwidth at a loop performance of  $P_{\text{peak}} B_m / B_c \approx 2.3 \text{ GF/s}$ . However, the single-threaded triad benchmark only achieves about 940 MF/s, which corresponds to a bandwidth of 15 GB/s (Fortran version, compiler options `-O3 -openmp -xAVX -opt-streaming-stores never -nolib-inline`).

The ECM model attributes this fact to non-overlapping contributions from core execution and data transfers: While loads and stores to the three arrays are accessing the L1 cache, no refills or evicts between L1 and L2 can occur. The same may be true for the lower cache levels and even memory, so that memory bandwidth is not the sole performance-limiting factor anymore. Core execution and transfers between higher cache levels are not completely hidden and the maximum memory bandwidth cannot be hit. However, when multiple cores access main memory (or a lower cache level with a bandwidth bottleneck, like the L3 cache of the Intel Westmere), the associated core times and data delays can overlap among the cores, and there will be a point where the bottleneck becomes relevant. Thus, it is possible to predict when performance saturation sets in.

### 3.2 The ECM model: Multicore scaling

The single-core ECM model predicts lower and upper limits for the bandwidth pressure on all memory hierarchy levels. When the bandwidth capacity of one level is exhausted, performance starts to saturate [4]. On the Intel Sandy Bridge processor, where the only shared bandwidth resource is the main memory interface, this happens when the bandwidth pressure exceeds the practical limit as measured, e.g., by a suitable multi-threaded STREAM-like benchmark. We call this the “saturation point.” At this point, the performance prediction from a balance model (see above) works well.

The maximum main memory bandwidth is an input parameter for the model. In principle it is possible to use the known parameters of the memory interface and the DIMM configuration, but this is over-optimistic in practice. For current Intel processors, the memory bandwidth achievable with standard streaming benchmarks like the McCalpin STREAM [19, 20] is between 65 and 85% of the theoretical maximum. Architectural peculiarities, however, may then impede optimal use of the memory interface with certain types of code. One example are data streaming loops with a very large number of concurrent load/store streams, which appear, e.g., in implementations of the lattice-Boltzmann algorithm (see Sect. 5 below). The full memory bandwidth as seen with the STREAM benchmarks cannot be achieved under such conditions. The reasons for this failure are unknown to us and will be further investigated.

### 3.3 Validation via streaming benchmarks

We validate the ECM model by using the Schönauer vector triad [1] as a throughput benchmark (see Listing 1) on the Sandy Bridge architecture. The Schönauer triad is similar to the STREAM triad but has one additional load stream. Note that there is no real work sharing in the benchmark loop (lines 13–15), since the purpose of the code is to fathom the bottlenecks of the architecture. The code is equipped with Intel compiler directives to

Listing 1: Pseudo-code for the vector triad throughput benchmark, including performance measurement. The actual benchmark loop is highlighted.

---

```

double precision, dimension(:), allocatable :: A,B,C,D
! Intel-specific: 512-byte alignment of allocatables
!DEC$ ATTRIBUTES ALIGN: 512 :: A,B,C,D

5   call get_walltime(S)

!$OMP PARALLEL PRIVATE(A,B,C,D,i,j)
...
    do j=1,R
10  ! Intel-specific: Assume aligned moves
    !DEC$ vector aligned
    !DEC$ vector temporal
        do i=1,N
            A(i) = B(i) + C(i) * D(i)
15          enddo
        ! prevent loop interchange
        if(A(N/2).lt.0) call dummy(A,B,C,D)
        enddo
    !$OMP END PARALLEL
20
    call get_walltime(E)

    WT = E-S

```

---

point out some crucial choices: All array accesses are aligned to suitable address boundaries (lines 3 and 11) to allow for aligned MOV instructions, which are faster on some architectures. Furthermore, the generation of nontemporal store instructions (“streaming stores”) is prevented (line 12).

### 3.3.1 Single-core analysis

All loop iterations are independent. Six full-width AVX loads and two full-width AVX stores are required to execute one unit of work (eight scalar iterations, or sixteen flops, respectively). From the microarchitectural properties described in Sect. 2.1 we know that this takes six cycles, since the stores can be overlapped with the loads. In Fig. 1 this is denoted by the arrows between the L1D cache and the registers. The floating-point instructions do not constitute a bottleneck, because only two ADDs and two MULTs are needed. Neglecting the loop counter and branch “mechanics,” which is justified because it can be reduced by inner loop unrolling and handled by other execution ports concurrently, the code has an overall instruction throughput of two  $\mu\text{ops}$  per cycle (six being the limit). In conclusion, the in-core performance is limited by load/store throughput, and we have  $T_{\text{core}} = 6\text{cy}$ .

Due to the write misses on array  $A(:)$ , an additional cache line load has to be taken into account whenever the data does not fit into the L1 cache. This is indicated by the red arrows in Fig. 1. Ten cycles each are needed for the data transfers between L2 and L1, and between L3 and L2, respectively. As mentioned above, the memory bandwidth limit of 36 GB/s leads to a per-cycle effective transfer width of 107 bits, which adds another 24 cy to  $T_{\text{data}}$ .

Figure 2 shows how the different parts can be put together to arrive at an estimate for the execution time. In the worst case, the contributions to  $T_{\text{data}}$  can neither overlap with each other nor with  $T_{\text{core}}$ , leading to  $T = 50\text{cy}$  for data in memory, 26cy for L3, and 16cy for L2 (see Fig. 2a). On the other hand, assuming full overlap beyond the L2 cache (see Fig. 2c), the minimum possible execution times are  $T = 24\text{cy}$ , 16cy, and 16cy, respectively. The only well-known fact in terms of overlap is that the L1 cache is single-ported, which is why no overlap is assumed even in the latter case.

Assuming the non-overlap condition for all cache levels, we arrive at the situation depicted in Fig. 2b: Contributions can only overlap if they involve a mutually exclusive set of caches. We then arrive at a prediction of  $T = 34\text{cy}$  for in-memory data, 20cy for L3, and again 16cy for L2 (the latter cannot be shown in the figure).

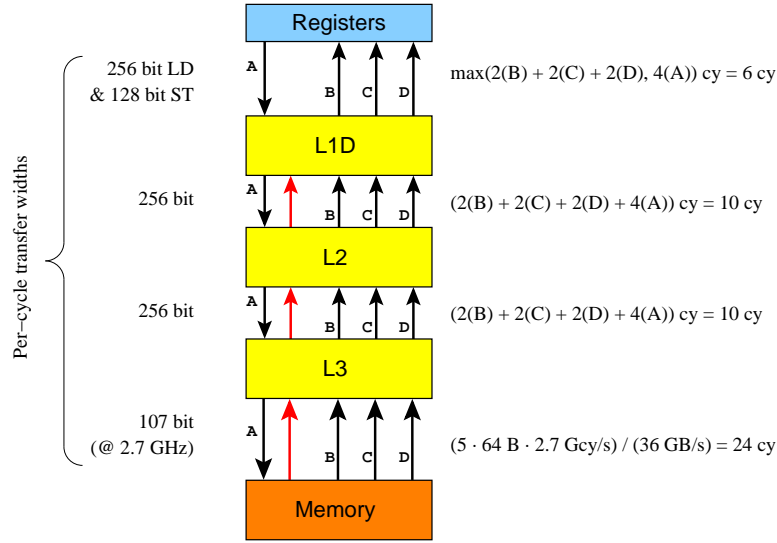


Figure 1: Single-core ECM model for the Schönauer triad benchmark ( $A(:)=B(:)+C(:)*D(:)$ ) on Sandy Bridge. The indicated cycle counts refer to eight loop iterations, i.e., a full cache line length per stream. The transfer width per cycle for refills from memory to L3 is derived from the measured STREAM bandwidth limit of 36 GB/s. Red arrows indicate write-allocate transfers.

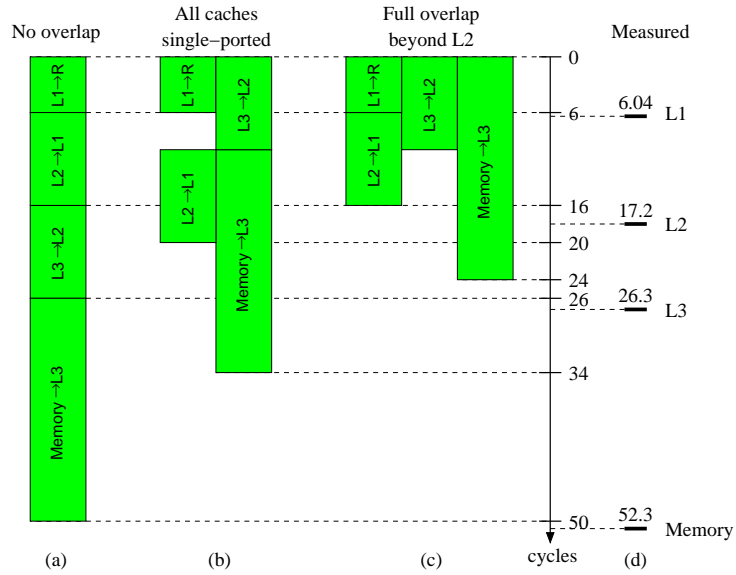


Figure 2: (a)–(c): Single-core cycle estimates for the Schönauer triad benchmark on Sandy Bridge, with different overlap assumptions. (d): Measurement at the base clock frequency of 2.7 GHz. See text for full details.



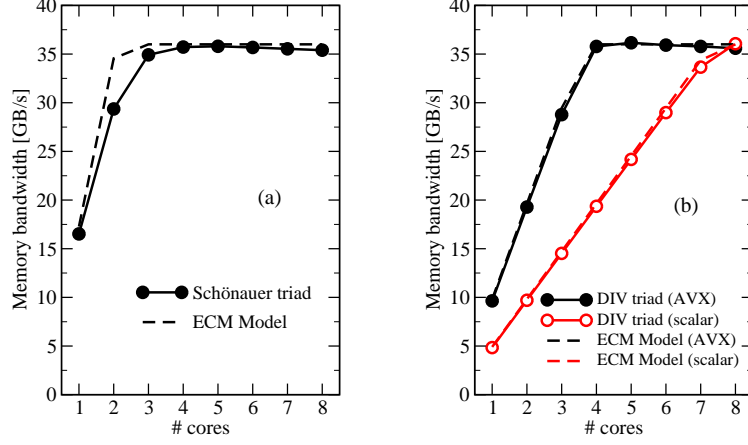


Figure 3: Multicore scaling of (a) the memory-bound Schönauer triad benchmark and (b) the modified triad with a divide ( $A(:) = B(:) + C(:) / D(:)$ ), in comparison with the corresponding ECM models (dashed lines) on a 2.7 GHz Sandy Bridge chip. The model for (a) assumes no overlap, while the model in (b) assumes full overlap of  $T_{\text{core}}$  with  $T_{\text{data}}$ .

Figure 2d shows measured execution times for comparison. We must conclude that there is no overlap taking place between any contributions to  $T_{\text{core}}$  and  $T_{\text{data}}$ . Note that this analysis is valid for a single type of processor, and that other microarchitectures may show different behavior. It must also be stressed that the existence of overlap also depends strongly on the type of code (see also the next section).

### 3.3.2 Multicore scaling

All resources in the Sandy Bridge processor chip, except for the memory interface, scale with the number of cores. Hence we predict good scalability of the benchmark loop up to eight cores if the data resides in the L3 cache, and indeed we see a speedup of 7.4 from one to eight cores. In the memory-bound regime we expect scalability up to the bandwidth limit of 107 bits/cy, which is a factor of 2.09 larger than the single-core bandwidth prediction of 320 bytes/50 cy = 51.2 bits/cy. The performance of the Schönauer triad loop should thus saturate at three cores, with a small speedup from two to three. Figure 3 shows a comparison of the model with measurements on a Sandy Bridge chip at 2.7 GHz. The model tracks the overall scaling behavior well, especially the point where saturation sets in. Note that we expect the same general characteristics for all loop kernels that are strongly load/store-bound in the L1 cache if the data traffic volume between all cache levels is roughly constant.

For comparison, we modified the vector triad code so that a divide is executed instead of a multiplication between arrays  $C(:)$  and  $D(:)$ . The throughput of the double-precision full-width AVX divide on the Sandy Bridge microarchitecture is 44 cycles if no shortcuts can be taken by the hardware [13], while the throughput of a scalar divide is 22 cycles. All required loads and stores in the L1 cache can certainly be overlapped with the large-latency divides, leading to an in-core execution time of 88 cy and 172 cy, respectively, for the AVX and scalar variants. In this case, the single-portedness of the L1 cache is not applicable, since the in-core code is not load/store-bound. Even if no overlap takes place in the rest of the hierarchy, the  $10 + 10 + 24 = 44$  additional cycles for data transfers (see Fig. 1) can be hidden behind the in-core time. The results in Fig. 3b show a very good agreement of the ECM model with the measurements.

## 4 Power dissipation and performance on multicore

In this section we will first investigate the power dissipation properties of the Sandy Bridge processor by studying several benchmark codes. We then develop a simple power model and derive from it the most interesting features for the “energy to solution” metric with respect to clock frequency, number of cores utilized, and serial code performance.

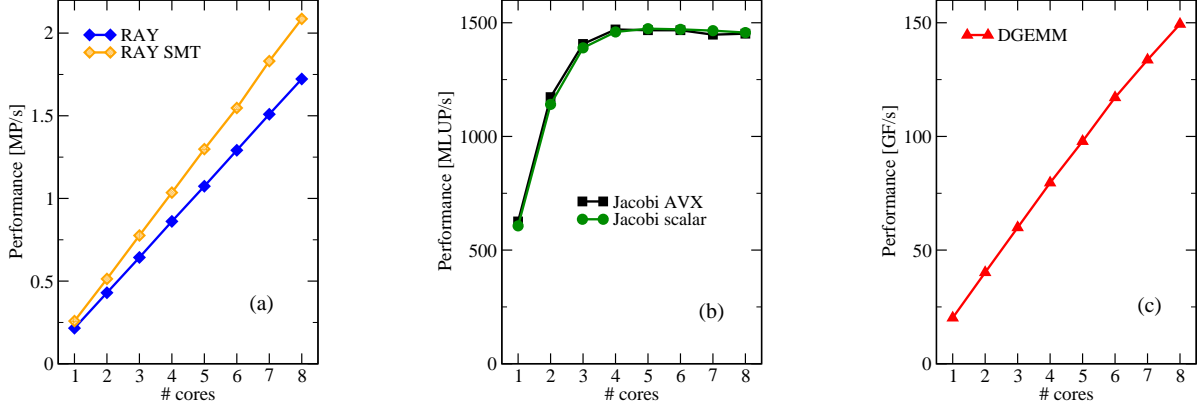


Figure 4: Performance of the benchmark codes on a Sandy Bridge chip with respect to the number of active cores at the base frequency of 2.7 GHz.

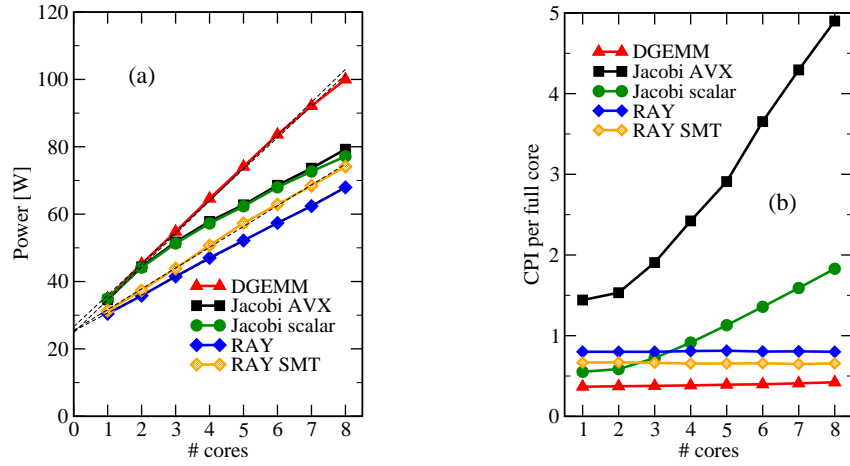


Figure 5: (a) Power dissipation and (b) cycles per instruction of the benchmark codes with respect to the number of used cores.

## 4.1 Power and performance behavior of benchmarks

A couple of test codes was chosen, each of which shows a somewhat typical behavior for a certain class of applications. We report performance, CPI (cycles per instruction), and power dissipation on a Sandy Bridge EP (Xeon E5-2680) chip, and track the dependence of power dissipation on clock frequency. The “turbo mode” feature was deliberately ignored.

In the following we briefly describe the benchmark codes together with performance and power data with respect to the number of used cores (see Figs. 4 and 5).

### 4.1.1 RAY

is a small, MPI-parallel, master-worker style ray-tracing program, which computes an image of  $15000^2$  gray-scale pixels of a scene containing several reflective spheres. Performance is reported in million pixels per second (MP/s).

Scalability across the cores of a multicore chip is perfect (see Fig 4a) since all data comes from L1 cache, load imbalance is prevented by dynamic work distribution, and there is no synchronization apart from infrequent communication of computed tiles with the master process, which is pinned to another socket and thus not taken



into account in the analysis below. The code is purely scalar and shows a mediocre utilization of the core resources with a CPI value of about 0.8 (see Fig 5b). It benefits to some extent from the use of simultaneous multi-threading (SMT), which reduces the CPI to 0.65 per (full) core for a speedup of roughly 15%. At the same time, power dissipation grows by about 8% and is roughly linear in the number of used cores for both cases (see Fig. 5a).

#### 4.1.2 Jacobi

is an OpenMP-parallel 2D Jacobi smoother (four-point stencil) used with an out-of-cache data set ( $4000^2$  lattice sites at double precision). Being load/store-dominated with an effective code balance of 6 B/F [21], it shows the typical saturation behavior described by the ECM model for streaming codes. Performance is reported in million lattice site updates per second (MLUP/s), where one update comprises four flops. Hence, we expect a saturation performance of 6 GF/s or 1500 MLUP/s on a full Sandy Bridge chip, which is fully in line with the measurement (see Fig. 4b).

This benchmark was built in two variants, an AVX-vectorized version and a scalar version, to see the influence on data-parallel instructions on power dissipation. Both versions have very similar scaling characteristics, with the scalar code being slightly slower below the saturation point, as expected. The performance saturation is also reflected in the CPI rate (Fig. 5b), which shows a linear slope after saturation. Surprisingly, although there is a factor of 2.5 in terms of CPI between the scalar and AVX versions, the power dissipation hardly changes (Fig. 5a). Beyond the saturation point, the slope of the power dissipation decreases slightly, indicating that a large CPI value is correlated with lower power. However, the relation is by no means inversely proportional, just as for the RAY benchmark.

#### 4.1.3 DGEMM

performs a number of multiplications between two dense double precision matrices of size  $5600^2$ , using the thread-parallel Intel MKL library that comes with the Intel compiler (version 10.3 update 9). Performance is reported in GF/s.

The code scales almost perfectly with a speedup of 7.5 on eight cores, and reaches about 86% of the arithmetic peak performance on the full Sandy Bridge chip at a CPI of about 0.4 (i.e, 2.5 instructions per cycle). Power dissipation is almost linear in the number of used cores (Fig. 5a).

DGEMM achieves the highest power dissipation of all codes considered here. Note that at the base frequency of 2.7 GHz, the thermal design power (TDP) of the chip of 130 W is not nearly reached, not even with the DGEMM code. With turbo mode enabled (3.1 GHz at eight cores) we have measured a maximum sustained power of 122 W. The Sandy Bridge chip can exceed the TDP limit for short time periods [11], but this was not investigated here.

Surprisingly, the power dissipation of DGEMM is identical to the Jacobi code (scalar and AVX versions) as long as the latter is not bandwidth-bound, whereas the RAY benchmark draws about 15% less power at low core counts. We attribute this to the mediocre utilization of the execution units in RAY, where some long-latency floating-point divides incur pipeline stalls, and the strong utilization of the full cache hierarchy by the Jacobi smoother.

#### 4.1.4 Power and performance vs. clock frequency

Figure 6a shows the power dissipation of all codes with respect to the clock frequency ( $f = 1.2 \dots 2.7$  GHz) when all cores are used (all virtual cores in case of the SMT variant of RAY). The solid lines are least-squares fits to a second-degree polynomial,

$$W(f) = W_0 + w_1 f + w_2 f^2, \quad (1)$$

for which the coefficient of the linear term is very small compared to the constant and the quadratic term. The quality of the fit suggests that the dependence of dynamic power dissipation on frequency is predominantly quadratic with  $7 \text{ W/GHz}^2 < w_2 < 10 \text{ W/GHz}^2$ , depending on code characteristics. The “base power”  $W_0 \approx 23 \text{ W}$  (leakage) is largely independent of the type of code, which is plausible; however, one should note that an extrapolation to  $f = 0$  is problematic here, so that the estimate for  $W_0$  is very rough.

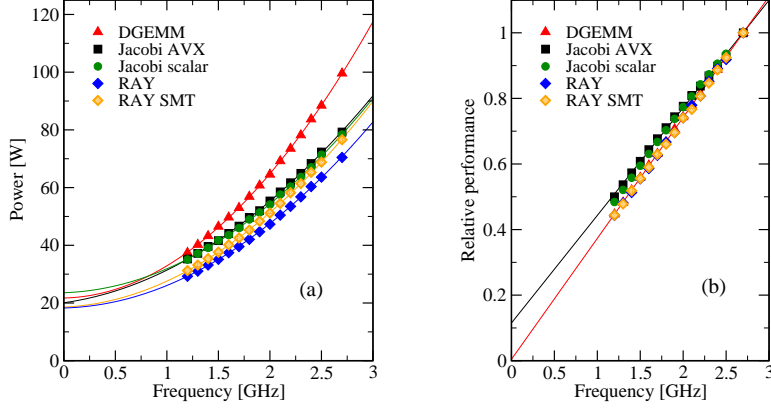


Figure 6: (a) Power dissipation of a Sandy Bridge chip with respect to clock speed for the benchmark codes. All eight physical cores were used in all cases, and all 16 virtual cores for the “RAY SMT” benchmark. The solid lines are least-squares fits to a second-degree polynomial in  $f$ . (b) Relative performance versus clock speed with respect to the 2.7 GHz level of single-core execution for the benchmark codes. Two processes on one physical core were used in case of RAY SMT. The solid lines are linear fits to the Jacobi AVX and DGEMM data, respectively.

In Fig. 6b we show the single-core performance of all benchmarks with respect to clock frequency, normalized to the level at  $f = 2.7$  GHz. As expected, the codes with near-perfect scaling behavior across cores show a strict proportionality of performance and clock speed, since all required resources run with the core frequency. In case of the Jacobi benchmark the linear extrapolation to  $f = 0$  has a small y-intercept, since resources are involved that are clocked independently of the CPU cores. The ECM model predicts this behavior if one can assume that the maximum bandwidth of the memory interface is constant with varying frequency.

Figure 7 shows the saturated memory bandwidth of a Sandy Bridge chip with respect to clock speed. If we assume that the core frequency should not influence the memory interface, we have no explanation for the drop in bandwidth below about 1.7 GHz (the ECM model predicts constant bandwidth for a streaming kernel like, e.g., the Schönauer triad). For the purpose of developing a multicore power model we neglect these effects and assume a strictly linear behavior (with zero y-intercept) of performance vs. clock speed in the non-saturated case.

#### 4.1.5 Conclusions from the benchmark data

In order to arrive at a qualitative model that connects the power and performance features of the multicore chip, we draw some general, simplifying conclusions from the data that was discussed in the previous sections:

- The dynamic power dissipation is quadratic in the clock frequency and parameterized by  $w_2$  in (1).  $w_2$  depends strongly on the type of code executed, and there is some (inverse) correlation with the CPI value, but a simple mathematical relation cannot be derived. The linear part  $w_1$  is generally small compared to  $w_2$ .
- A linear extrapolation of power dissipation vs. the number of active cores (dashed lines in Fig. 5a) to zero cores shows that the baseline power of the chip is  $W_0 \approx 25$  W, independent of the type of running code. In case of the bandwidth-limited Jacobi benchmark only the one- and two-core data points were considered in the extrapolation. The result for  $W_0$  is also in line with the quadratic extrapolations to zero clock frequency in Fig. 6a.
- The dependence of power dissipation on the number of active cores  $t$  is linear in the non-saturated regime,

$$W(f, t) = W_0 + (W_1 f + W_2 f^2) t, \quad (2)$$

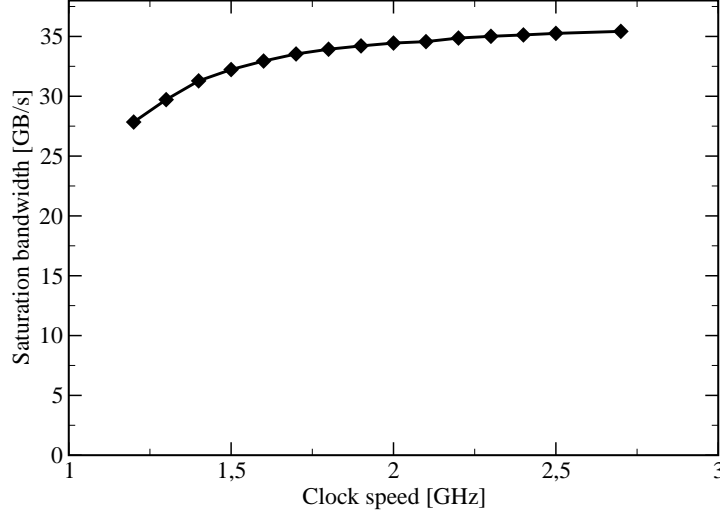


Figure 7: Maximum memory bandwidth (saturated) versus clock frequency of a Sandy Bridge chip.

so that  $w_{1/2} = t \cdot W_{1/2}$ . Although the power per core rises more slowly in the saturation regime, we regard this as a second-order effect and neglect it in the following: The fact that a core is active has much more influence on power dissipation than the characteristics of the running code.

- In the non-saturated case performance is proportional to the clock speed.
- Using both hardware threads (virtual cores) on a physical Sandy Bridge core increases power dissipation due to the improved utilization of the pipelines. The corresponding performance increase depends on the code, of course, so it may be more power-efficient to ignore the SMT threads. In case of the RAY code, however, the increase in power is over-compensated by a larger boost in performance. See Sect. 4.3 for details.

## 4.2 A qualitative power model

Using the measurements and conclusions from the previous section we can now derive a simple power model that describes the overall power properties of a multicore chip with respect to number of cores used, the scaling properties of the code under consideration, and the clock frequency. As a starting point we choose the “energy to solution” metric, which quantifies the energy required to solve a certain compute problem. Hence, we restrict ourselves to strong scaling scenarios. This is not a severe limitation, since weak scaling is usually applied in the massively (distributed-memory) parallel case, where the relevant scaling unit is a node or a ccNUMA domain (which is usually a chip). The optimal choice of resources and execution parameters on the chip level, where the pertinent bottlenecks are different, are done at a constant problem size.

The following basic assumptions go into the model:

1. The whole  $N_c$ -core chip consumes some baseline power  $W_0$  when powered on, which is independent of the number of active cores and of the clock speed.
2. An active core consumes a dynamic power of  $W_1 f + W_2 f^2$ . We will also consider deviations from some baseline clock frequency  $f_0$  such that  $f = (1 + \Delta v) f_0$ , with  $\Delta v = \Delta f / f_0$ .
3. At the baseline clock frequency, the serial code under consideration runs at some performance  $P_0$ . As long as there is no bottleneck, the performance is linear in the number of used cores  $t$  and the normalized clock speed,  $1 + \Delta v$ . The latter dependence will not be exactly linear if some part of the hardware (e.g., the

outer-level cache) runs at its own clock speed. In presence of a bottleneck (like, e.g., memory bandwidth), the overall performance with respect to  $t$  is capped by some maximum value  $P_{\max}$ :

$$P(t) = \min((1 + \Delta v)tP_0, P_{\max}) . \quad (3)$$

Here we assume that performance scales with clock frequency until  $P_{\max}$  is reached.

Since time to solution is inverse performance, the energy to solution becomes

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{\min((1 + \Delta v)tP_0, P_{\max})} . \quad (4)$$

#### 4.2.1 Minimum energy with respect to number of used cores

Due to the assumed saturation of performance with  $t$ , we have to distinguish two cases:

**Case 1:**  $(1 + \Delta v)tP_0 < P_{\max}$  Performance is linear in the number of cores, so that (4) becomes

$$E = \frac{W_0 + (W_1 f + W_2 f^2)t}{(1 + \Delta v)tP_0} , \quad (5)$$

and  $E$  is a decreasing function of  $t$ :

$$\frac{\partial E}{\partial t} = -\frac{W_0}{(1 + \Delta v)t^2 P_0} < 0 . \quad (6)$$

Hence, the more cores are used, the shorter the execution time and the smaller the energy to solution.

**Case 2:**  $(1 + \Delta v)tP_0 > P_{\max}$  Performance is constant in the number of cores, hence

$$E = \frac{1}{P_{\max}} (W_0 + (W_1 f + W_2 f^2)t) \quad (7)$$

$$\Rightarrow \frac{\partial E}{\partial t} = \frac{1}{P_{\max}} (W_1 f + W_2 f^2) > 0 . \quad (8)$$

In this case, energy to solution grows with  $t$ , with a slope that is proportional to the dynamic power, while the time to solution stays constant; using more cores is thus a waste of energy.

For codes that show performance saturation at some  $t_s$ , it follows that energy (and time) to solution is minimal just at this point:

$$t_s = \frac{P_{\max}}{(1 + \Delta v)P_0} . \quad (9)$$

If the code scales to the available number of cores, case 1 applies and one should use them all.

#### 4.2.2 Minimum energy with respect to serial code performance

Since the serial code performance  $P_0$  only appears in the denominator of (4), increasing  $P_0$  leads to decreasing energy to solution unless  $P = P_{\max}$ . A typical example for this scenario is the SIMD vectorization of a bandwidth-bound code: Using data-parallel instructions (such as SSE or AVX) will generally reduce the execution overhead in the core, so that  $P_0$  grows and the saturation point  $P_{\max}$  is reached at smaller  $t$  (see (9)). Consequently, the potential for saving energy is twofold: When operating below the saturation point, optimized code requires less energy to solution. At the saturation point, one can get away with fewer active cores to solve the problem at maximum performance.

#### 4.2.3 Minimum energy with respect to clock frequency

We again have to distinguish two cases:

**Case 1:**  $(1 + \Delta v)tP_0 < P_{\max}$  Energy to solution is the same as in (5) and  $f = (1 + \Delta v)f_0$ , so that

$$\frac{\partial E}{\partial \Delta v} = \frac{f_0^2}{tP_0} \left( W_2 t - \frac{W_0}{f^2} \right). \quad (10)$$

The derivative is positive for large  $f$ ; setting it to zero and solving for  $f$  thus yields the frequency for minimal energy to solution:

$$f_{\text{opt}} = \sqrt{\frac{W_0}{W_2 t}}. \quad (11)$$

A large baseline power  $W_0$  forces a large clock frequency to “get it over with” (“clock race to idle”). Depending on  $W_0$  and  $W_2$ ,  $f_{\text{opt}}$  may be larger than the highest possible clock speed of the chip, so that there is no energy minimum. This may be the case if one includes the rest of the system into the analysis (i.e., memory, disks, etc.). On the other hand, a large dynamic power  $W_2$  allows for smaller  $f_{\text{opt}}$ , since the loss in performance is over-compensated by the reduction in power dissipation. The fact that  $f_{\text{opt}}$  does not depend on  $W_1$  just reflects our assumption that the serial performance is linear in  $f$ .

Since  $t$  appears in the denominator in (11), it is tempting to conclude that a clock frequency reduction can be compensated by using more cores, but the influence on  $E$  has to be checked by inserting  $f_{\text{opt}}$  from (11) into (5):

$$E(f_{\text{opt}}) = \frac{f_0}{P_0} \left( 2\sqrt{\frac{W_0 W_2}{t}} + W_1 \right) \quad (12)$$

This confirms our conjecture that, below the saturation point, more cores at lower frequency save energy. At the same time, performance at  $f_{\text{opt}}$  is

$$P(f_{\text{opt}}) = \frac{f_{\text{opt}}}{f_0} t P_0 = \frac{P_0}{f_0} \sqrt{\frac{W_0 t}{W_2}}, \quad (13)$$

hence it grows with the number of cores: trading cores for clock slowdown does not compromise time to solution.

However, if  $t$  is fixed, (13) also tells us that, if  $f_{\text{opt}} < f_0$ , performance will be smaller than at the base frequency  $f_0$ , although the energy to solution is also smaller. This may be problematic if  $t$  cannot be made larger to compensate for the loss in performance. In this case the energy to solution metric is insufficient and one has to choose a more appropriate cost function, such as energy multiplied by runtime:

$$C = \frac{E}{P} = \frac{W_0 + (W_1 f + W_2 f^2)t}{((1 + \Delta v)tP_0)^2}. \quad (14)$$

Differentiating  $C$  with respect to  $\Delta v$  gives

$$\frac{\partial C}{\partial \Delta v} = -\frac{2W_0 + W_1 f t}{(f/f_0)^3 P_0^2} < 0. \quad (15)$$

Hence, a higher clock speed is always better if  $C$  is chosen as the relevant cost function.

**Case 2:**  $(1 + \Delta v)tP_0 > P_{\max}$  Beyond the saturation point, energy to solution is the same as in (7), so it grows with the frequency: The clock should be as slow as possible. Together with the findings from case 1 this means that minimal energy to solution is achieved when using all available cores, at a clock frequency which is so low (if possible) that the saturation point is right at  $t = N_c$ .

These results reflect the popular “clock race to idle” idiom, which basically states that a processor should run at maximum frequency to “get it over with” and go to sleep as early as possible to eventually save energy. Using the energy to solution behavior as derived above, we now know how this strategy depends on the number of cores used and the ratio of baseline and dynamic power. “Clock race to idle” makes sense only in the sub-saturation regime, and when  $f < f_{\text{opt}}$ . Beyond  $f_{\text{opt}}$  (if such frequencies are allowed), the quadratic dependence of power on clock speed will waste energy. Beyond the saturation point, i.e., if  $t > t_s$ , lower frequency is always better.

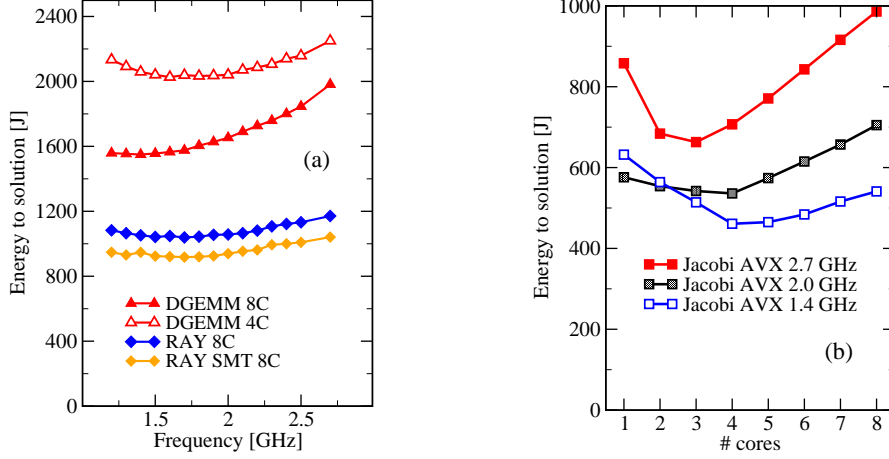


Figure 8: Energy to solution for (a) the scalable benchmarks DGEMM (eight and four cores) and RAY (eight cores) versus clock frequency on a Sandy Bridge socket and (b) the Jacobi AVX benchmark versus number of cores at different core frequencies.

### 4.3 Energy to solution for benchmarks

Figure 8 shows energy to solution measurements for the scalable codes (Fig. 8a) and the Jacobi AVX benchmark (Fig. 8b) versus clock frequency and number of cores, respectively.

Comparing the frequency for minimum energy to solution between DGEMM and RAY at eight cores (solid symbols in Fig. 8a), we can identify the behavior predicted by (11): A large dynamic power factor  $W_2$  leads to lower  $f_{\text{opt}}$ . The SMT version of RAY consumes more power than the standard version, but, as anticipated above, the larger performance leads to lower energy to solution: Better resource utilization on the core, i.e., optimized code, saves energy; this provides another possible attitude towards the “race to idle” idiom. Given the huge amount of optimization potential that is still hidden in many production codes on highly parallel systems, we regard this view as even more relevant than optimizing clock speed for a few percent of energy savings.

Eq. (11) predicts a larger optimal frequency  $f_{\text{opt}}$  at fewer cores, which is clearly visible when comparing the four- and eight-core energy data for DGEMM in Fig. 8a (solid vs. open triangles). At the same time, fewer cores also lead to larger minimum energy to solution at  $f_{\text{opt}}$ , which was shown in (12).

The Jacobi benchmark shows all the expected features of a code whose performance saturates at a certain number of cores  $t_s$ : As predicted by the ECM model, the saturation point is shifted to a larger number of cores as the clock frequency goes down; we have derived in Sect. 4.2.1 that this is the point at which energy to solution is minimal. Lowering the frequency,  $t_s$  gets larger, but energy to solution decreases (see (12)). When  $t > t_s$ , more cores and higher clock speed both are a waste of energy. At  $t < t_s$  the Jacobi code is largely frequency-bound and there is an optimal frequency  $f_{\text{opt}} \approx 2 \text{ GHz}$  with minimal energy to solution. Here we substantiate the prediction from Sect. 4.2.3 that “clock race to idle” is largely counterproductive if we look at the chip’s power dissipation only. See also Sect. 5.3 for a discussion of “race to idle” in the context of a lattice-Boltzmann CFD solver.

In conclusion, although considerable simplifications have gone into the model (4), it is able to describe the qualitative behavior of the benchmark applications with respect to energy to solution well.

## 5 Case study: A D3Q19 lattice-Boltzmann fluid solver

### 5.1 The lattice-Boltzmann algorithm and its implementation

The widely used class of lattice Boltzmann models with BGK approximation of the collision process [22, 23, 24, 25] is based on the evolution equation

$$f_i(\vec{x} + \vec{e}_i \delta t, t + \delta t) = f_i(\vec{x}, t) - \frac{1}{\tau} [f_i(\vec{x}, t) - f_i^{\text{eq}}(\rho, \vec{u})] \quad i = 0 \dots N. \quad (16)$$

Here,  $f_i$  denotes the particle distribution function (pdf), which represents the fraction of particles located in timestep  $t$  at position  $\vec{x}$  and moving with the microscopic velocity  $\vec{e}_i$ . The relaxation time  $\tau$  determines the rate of approach to local equilibrium and is related to the kinematic viscosity of the fluid. The equilibrium state  $f_i^{\text{eq}}$  itself is a low Mach number approximation of the Maxwell-Boltzmann equilibrium distribution function. It depends only on the macroscopic values of the fluid density  $\rho$  and the flow velocity  $\vec{u}$ . Both can be easily obtained as the first moments of the particle distribution function. The discrete velocity vectors  $\vec{e}_i$  arise from the  $N$  chosen collocation points of the velocity-discrete Boltzmann equation and determine the basic structure of the numerical grid. A typical 3D discretization is the D3Q19 model [25], which uses 19 discrete velocities (collocation points).

Each timestep ( $t \rightarrow t + \delta t$ ) consists of the following steps, which are repeated for all cells of the computational domain:

- Compute the local macroscopic flow quantities  $\rho$  and  $\vec{u}$  from the distribution functions,  $\rho = \sum_{i=0}^N f_i$  and  $\vec{u} = \frac{1}{\rho} \sum_{i=0}^N f_i \vec{e}_i$ .
- Calculate the equilibrium distribution  $f_i^{\text{eq}}$  from the macroscopic flow quantities (see [25] for the equation and parameters) and execute the “collision” (relaxation) process,  $f_i^*(\vec{x}, t^*) = f_i(\vec{x}, t) - \frac{1}{\tau} [f_i(\vec{x}, t) - f_i^{\text{eq}}(\rho, \vec{u})]$ , where the superscript “\*” denotes the post-collision state (“collide step”).
- Propagate the  $i = 0 \dots N$  post-collision states  $f_i^*(\vec{x}, t^*)$  to the appropriate neighboring cells according to the direction of  $\vec{e}_i$ , resulting in  $f_i(\vec{x} + \vec{e}_i \delta t, t + \delta t)$ , i.e., the values of the next timestep (“stream step”).

The first two steps are computationally intensive and involve values of the local node only. They can easily be combined and will be denoted “collide step” in the following. The third step is a direction-dependent uniform shift of data in memory involving no floating-point computation and will be denoted “streaming step”. A fourth step, the so-called “bounce-back” rule [26], is incorporated as an additional part of the stream step and “reflects” the distribution functions at the interface between fluid and solid cells, resulting in an approximate no-slip boundary condition at walls. Since bounce-back is only required at the interfaces, it has a minor impact on runtime if the ratio of solid to fluid cells is small. Hence, we will omit it for the analysis below.

### 5.1.1 Combined Collide-Stream kernel

The LBM algorithm builds on the two physical processes, *collision* and *streaming*. For this report we use the so-called “pull” scheme, i.e., the streaming step is followed by the collision. Using appropriate data storage schemes, e.g., two disjoint arrays, it is therefore possible to do both steps within a single loop. This reduces the overall data transfer between main memory and CPU to one load, one store, and one write-allocate per lattice cell update and distribution function.

### 5.1.2 Data layout

Another basic decision is the choice of the data layout for the arrays holding the particle distribution functions. Here we consider a full matrix representation, where the pdf array covers the full simulation domain independent of the character of the cells (fluid vs. solid). Choosing rectangular 3D computational domains this requires a pdf array with at least four dimensions (three spatial coordinates  $(x, y, z)$  and the discretization stencil  $(Q = 0, \dots, 18)$ ). We only consider the structure-of-arrays (SoA) approach ( $\text{PDF}(x, y, z, Q)$ , using Fortran notation), as it provides maximum attainable performance without the need for spatial blocking [27, 28].

### 5.1.3 Loop splitting

The SoA implementation requires 19 load and 19 store streams, addressing completely different parts of the pdf array concurrently. The large number of competing store streams is problematic for the underlying memory architecture due to, e.g., the limited number of store buffers. Thus it is often beneficial to separate the store streams by updating a smaller number of directions (1–5) at a time. This can easily be implemented by having separate  $x$  (inner) loops for the different chunks of store streams and holding some intermediate results in small buffer arrays (which comes at the cost of additional cache traffic and instructions). Listing 2 shows the pseudo-code for a D3Q19 SoA split-store LBM kernel.



Listing 2: Pseudo-code for the SoA D3Q19 LBM “split” benchmark kernel, with split-off loops highlighted. The bounce-back rule is omitted.

---

```

double precision, dimension(:,:,:,:),allocatable :: PDF
double precision, dimension(:),allocatable :: loc_dens
double precision, dimension(:),allocatable :: ux,uy,uz
...
5 !OMP PARALLEL SHARED(A)
!OMP DO
...
do z=1,Nz
do y=1,Ny
10 do x=1,Nx
! Caculate density
loc_dens(x) = PDF(x,y,z,C,t)
+ PDF(x-1,y,z,E,t)
+ PDF(x,y-1,z,N,t)
15 + PDF(x,y+1,z+1,BS,t)

!calculate moments and store to ux,uy,uz
ux(x)= PDF(x-1,y,z,E,t)-PDF(x+1,y,z,W,t)
20 + PDF(x-1,y-1,z,NE,t) +PDF(x+1,y-1,z,NW,t)...

uy(x)= PDF(x,y-1,z,N,t)-PDF(x,y+1,z,S,t)
+ PDF(x-1,y-1,z,NE,t) -PDF(x-1,y+1,z,SE,t)...

25 uz(x)=...
enddo

! 19 store loops: tN = new time, t = old time
do x=1,Nx
! reuse ux,uy,uz and loc_dens
! Propagation to local node
PDF(x,y,z,C,tN) = PDF(x,y,z,C,t) *... ux(x)...
enddo

35 do x=1,Nx
! reuse ux,uy,uz and loc_dens
! Propagation to local node
PDF(x,y,z,E,tN) = PDF(x-1,y,z,E,t) *... ux(x)...
enddo

40 ! 17 more propagation loops here ...
...

enddo
45 enddo
!OMP END DO
!OMP END PARALLEL

```

---

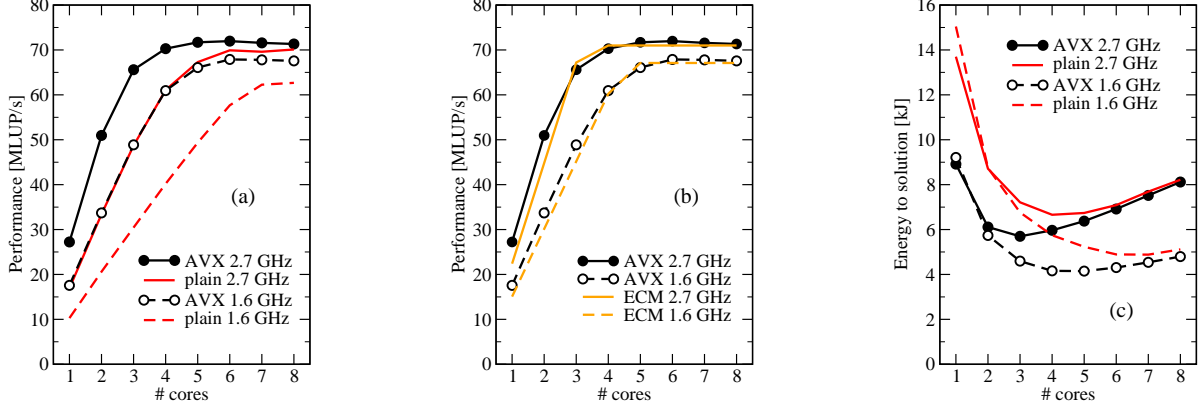


Figure 9: (a) Performance of the AVX and plain (scalar) variants of the LBM solver versus number of cores on a Sandy Bridge socket. (b) Comparison of the AVX LBM code with the non-overlap ECM model. (c) Energy to solution of the code variants from (a) versus number of cores.

### 5.1.4 SIMD processing

The optimizations of data layout and loop splitting allow for a large fraction of the maximum performance (in double precision) according to the roofline model. For single precision computation, however, performance does not double as would be expected. This is a clear indication for the need to improve the use of processor/on-chip resources by full vectorization. Considering the predictions of the ECM and power models with respect to single-threaded performance and energy to solution, respectively, SIMD vectorization will not only improve the single-precision performance but also, more importantly, shift the saturation point to a smaller number of cores, leading to lower energy to solution also in double precision. We have thus implemented a vectorized version of the LBM algorithm from Listing 2 using AVX SIMD intrinsics.

## 5.2 Performance results

We use a cubic domain of  $228^3$  fluid cells (about 3.7 GB of memory) without obstacles at double precision as a benchmark case for LBM. Figure 9a shows the performance of the plain (scalar) and AVX implementations of the LBM algorithm with respect to the number of cores at 2.7 GHz and 1.6 GHz, respectively. As expected, at 2.7 GHz the AVX version is faster below saturation and also saturates earlier, but arrives at the same maximum performance as the scalar code at larger core counts. At reduced clock speed the scalar code shows almost perfect scaling up to seven cores. The memory bandwidth degradation observed with simple streaming benchmarks (see Fig. 7) can also be seen here: About 5–10% of saturated performance is lost at 1.6 GHz, depending on the code variant.

## 5.3 Application of the performance model

The Intel Architecture Code Analyzer (IACA) [18] can compute pure execution time ( $T_{\text{core}}$ ) predictions from generated assembly code, using some built-in knowledge about the microarchitecture as input. It outputs several numbers: (i) The minimum number of cycles during which each of the core’s issue ports is busy, assuming no dependencies between instructions. The largest of those numbers (the “total throughput”) is the minimum execution time of the analyzed code in cycles. (ii) A refined prediction based on dependencies between instructions along the critical execution path (the “performance latency”). We generally use the total throughput in order to arrive at a prediction that is an absolute upper performance limit.

Listings 3 and 4 show the IACA analysis for the plain and AVX-based LBM kernel variants, respectively. The tool was given the complete assembly code for the statements between lines 10 and 43 in Listing 2. It predicts a minimum in-core execution time of 216 cy per AVX-vectorized LBM loop iteration, i.e., 432 cy per unit of work (eight lattice updates), the bottleneck being the LOAD ports. For the scalar code, the minimum prediction is

Listing 3: Intel Architecture Analyzer output for the AVX LBM kernel on Intel Sandy Bridge.

```

Analysis Report
-----
Total Throughput: 216 Cycles;          Throughput Bottleneck: Port2_DATA, Port3_DATA
Total number of Uops bound to ports: 656
5 Data Dependency Latency: 81 Cycles; Performance Latency: 271 Cycles

Port Binding in cycles:
-----
10 | Port | 0 - DV | 1 | 2 - D | 3 - D | 4 | 5 |
   | Cycles | 98 | 0 | 114 | 141 | 216 | 180 | 216 | 50 | 97 |
   -----

```

Listing 4: Intel Architecture Analyzer output for the scalar LBM kernel on Intel Sandy Bridge.

```

Analysis Report
-----
Total Throughput: 223 Cycles;          Throughput Bottleneck: Port1
Total number of Uops bound to ports: 899
5 Data Dependency Latency: 122 Cycles; Performance Latency: 259 Cycles

Port Binding in cycles:
-----
10 | Port | 0 - DV | 1 | 2 - D | 3 - D | 4 | 5 |
   | Cycles | 122 | 0 | 223 | 199 | 168 | 199 | 168 | 62 | 94 |
   -----

```

223 cy, i.e., 1784 cy per unit of work, with the ADD port as the bottleneck. The fact that the AVX prediction is almost exactly four times faster than the scalar one is pure coincidence, however: Inspection of the assembly code revealed that in some cases the Intel compiler fails to generate full-width AVX loads and stores from SIMD intrinsics but chooses half-width (128 bit) variants instead, especially for the write-back loops starting at line 29 in Listing 2. This is also the reason why the AVX code is limited by the LOAD port, whereas the scalar code is limited by the ADD port.

The inner loops require 19 load and store streams to the outer memory hierarchy. The problem size in  $x$  direction was chosen such that the loads and stores from and to the auxiliary arrays `loc_dens(:)`, `ux(:)`, `uy(:)`, and `uz(:)` are handled within the L1 cache. From a data traffic point of view, the LBM thus resembles 19 parallel, contiguous array copy operations. As already mentioned in Sect. 3.2, under such conditions the memory interface of the Sandy Bridge chip fails to achieve its maximum bandwidth according to the STREAM benchmarks, but saturates at about 32.3 GB/s at 2.7 GHz clock speed (30.6 GB/s at 1.6 GHz). We use these values as upper limits in the following.

Together with the IACA analysis we arrive at the single-core cycle counts displayed in Fig. 10 (we only discuss the more interesting AVX variant here, since the scalar code is strongly dominated by in-core execution). Assuming that no overlap takes place within the hierarchy, 965 cy (851 cy) are needed for one unit of work, which amounts to a performance of 22.4 MLUP/s (15 MLUP/s) at 2.7 GHz (1.6 GHz). Figure 9b shows a comparison of the model with measurements. At both frequencies the scaling behavior is tracked quite accurately, including the saturation point, but obviously the no-overlap assumption is slightly too pessimistic for this code and there must be some (small) overlap, most probably between execution and data transfers.

We will see in the next section that the accurate prediction of the saturation point has significance for the power behavior.

## 5.4 Application of the power model

Figure 9c shows the energy to solution for a fixed number of LBM updates versus the number of used cores. The general behavior is very similar to the Jacobi benchmark (see Fig. 8), and exactly as predicted by the power model for bandwidth-bound codes: Minimum energy to solution is achieved at the saturation point  $t = t_s$  predicted by the ECM model (e.g., at three to four cores for AVX at 2.7 GHz and at about five cores at 1.6 GHz). Using more cores at lower frequency has the drawback, however, that about 5–10 % of saturated performance is lost; a refined cost model similar to (14) should be used in this case to determine whether the loss in performance is outweighed by the gain in energy efficiency.

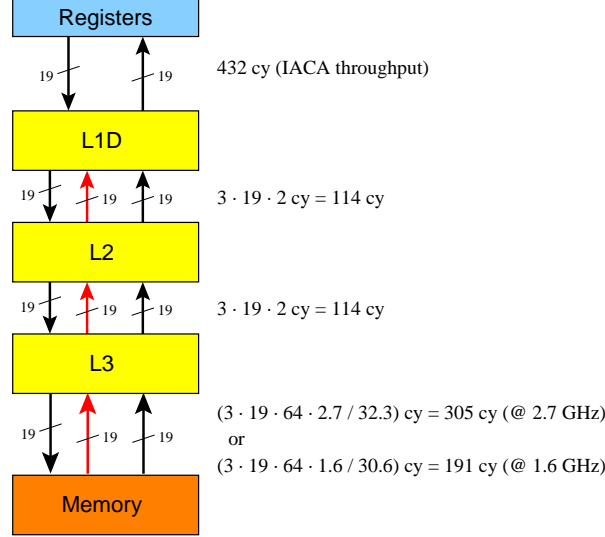


Figure 10: Single-core ECM model for the lattice-Boltzmann kernel with AVX on Sandy Bridge at 2.7 and 1.6GHz. One arrow represents the transfer of 19 cache lines for the 19 distribution functions. Red arrows indicate write-allocate transfers. The two predictions for the memory transfers emerge from the limit given by a multi-stream copy benchmark (32.3 and 30.6GB/s, respectively).

If there is no clear saturation (as with the plain variant at 1.6GHz), the available number of cores is barely sufficient to reach an energy minimum. In the serial case  $t = 1$ , where the chip’s power consumption is dominated by the baseline power  $W_0$ , a slow clock is counterproductive and leads to higher energy to solution (dashed lines above solid lines). This is the point where “clock race to idle” makes sense for this code, but using less than  $t_s$  cores would be extremely wasteful since energy to solution is more than twice as large as at the saturation point.

Finally, comparing the AVX and plain versions of the LBM implementation at the saturation point, we recover the obvious result that optimizing code saves energy: “Racing to idle” with better code is always beneficial.

## 6 Conclusion

### Summary

We have applied the Execution-Cache-Memory (ECM) model to simple streaming benchmarks and to a lattice-Boltzmann flow solver to show how the scaling properties of bandwidth-bound codes on a multicore chip can be described better than with a simple bottleneck analysis. Although our analysis on the Sandy Bridge EP processor suggests that there is no overlap between data transfers on different levels of the memory hierarchy, this may not be a general result and must be investigated on other microarchitectures and for more loop kernels.

Starting from measured performance and power dissipation properties of three benchmarks with different demands on the hardware we have furthermore established a simple model for “energy to solution” of parallel codes on a multicore chip. The model also takes typical saturation properties for bandwidth-bound loop kernels into account and thus builds implicitly on the ECM performance model. The behavior of energy to solution is qualitatively described by the model with respect to the number of active cores, the performance of the single-threaded code, and the clock frequency, despite substantial simplifications from the underlying measurements: We have derived a definite condition for the usefulness of “clock race to idle,” i.e., running with a high clock frequency to complete a computation as fast as possible, which applies when the code scales well with the number of cores. If there is a saturation point with respect to core count, lowest energy to solution is achieved when the number of threads is chosen so that saturation sets in, at the lowest possible clock speed. Finally, better code performance always improves energy to solution, and is thus the most important aspect in saving energy on

parallel systems. These results have been confirmed using the simple benchmarks on which the model was built, and by applying the model to a lattice-Boltzmann flow solver.

## Outlook

The specific issue of memory bandwidth degradation with loop kernels that have a large number of concurrent load/store streams is still poorly understood, and will be studied in more detail in the future. Future work will also encompass the application of the performance and power models, in order to fathom their domain of applicability, to more processor architectures and application codes and to massively parallel applications. We plan to extend the models to codes that exhibit more complex performance patterns such as load imbalance and synchronization or communication overhead.

We also expect future processor and system designs to make power information more accessible so that comprehensive power measurement functions can be implemented in the `likwid-perfctr` tool and used in analysis.

## Acknowledgements

This work was supported by the German Federal Ministry for Education and Research (BMBF) under grants no. 01IH08003A (project SKALB) and 01IH11002A (project hpcADD), and by the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR) via the project OMI4papps. We are indebted to Intel Germany for providing a Sandy Bridge EP test platform through an early access program.

## References

- [1] Schönauer W. *Scientific Supercomputing: Architecture and Use of Shared and Distributed Memory Parallel Computers*. Self-edition, 2000. URL <http://www.rz.uni-karlsruhe.de/~rx03/book>.
- [2] Williams SW, Waterman A, Patterson DA. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. *Technical Report UCB/EECS-2008-134*, EECS Department, University of California, Berkeley Oct 2008. URL <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-134.html>.
- [3] Treibig J, Hager G. Introducing a performance model for bandwidth-limited loop kernels. *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science*, vol. 6067, Wyrzykowski R, Dongarra J, Karczewski K, Wasniewski J (eds.). Springer Berlin / Heidelberg, 2010; 615–624, doi:10.1007/978-3-642-14390-8\_64.
- [4] Suleman MA, Qureshi MK, Patt YN. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. *SIGARCH Comput. Archit. News* Mar 2008; **36**(1):277–286, doi:10.1145/1353534.1346317.
- [5] Hoisie A, Lubeck O, Wasserman HJ. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *Int. J. High Perform. Comp. Appl.* 2000; **14**:330–346, doi:10.1177/109434200001400405.
- [6] Nudd GR, Kerbyson DJ, Papaefstathiou E, Perry SC, Harper JS, Wilcox DV. Pace — A toolset for the performance prediction of parallel and distributed systems. *Int. J. High Perform. Comp. Appl.* 2000; **14**(3):228–251, doi:10.1177/109434200001400306.
- [7] Kerbyson DJ, Alme HJ, Hoisie A, Petrini F, Wasserman HJ, Gittings M. Predictive performance and scalability modeling of a large-scale application. *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, ACM: New York, NY, USA, 2001; 37–37, doi:10.1145/582034.582071.

- [8] Kerbyson DJ, Jones PW. A performance model of the Parallel Ocean Program. *Int. J. High Perform. Comp. Appl.* 2005; **19**:261–276, doi:10.1177/1094342005056114.
- [9] Horvath T, Skadron K. Multi-mode energy management for multi-tier server clusters. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, ACM: New York, NY, USA, 2008; 270–279, doi:10.1145/1454115.1454153.
- [10] Li D, de Supinski BR, Schulz M, Nikolopoulos DS, Cameron KW. Strategies for energy efficient resource management of hybrid programming models. *IEEE Transactions on Parallel and Distributed Systems* 2012; **99**(PrePrints), doi:10.1109/TPDS.2012.95.
- [11] Rotem E, Naveh A, Ananthakrishnan A, Rajwan D, Weissmann E. Power-management architecture of the Intel microarchitecture code-named Sandy Bridge. *IEEE Micro* 2012; **32**:20–27, doi:10.1109/MM.2012.12.
- [12] Hähnel M, Döbel B, Völz M, Härtig H. Measuring energy consumption for short code paths using RAPL. *Proc. GREENMETRICS'12, London, UK*, 2012.
- [13] Intel 64 and IA-32 architectures optimization reference manual April 2012. URL <http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf>.
- [14] Treibig J, Hager G, Wellein G. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. *PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, IEEE Computer Society: Los Alamitos, CA, USA, 2010; 207–216, doi:10.1109/ICPPW.2010.38.
- [15] LIKWID performance tools. URL <http://code.google.com/p/likwid>.
- [16] Treibig J, Hager G, Wellein G. likwid-bench: An extensible microbenchmarking platform for x86 multicore environments. *Tools for High Performance Computing 2011*, Resch M, et al. (eds.). Springer Berlin Heidelberg, 2012. To appear.
- [17] Treibig J, Hager G, Hofmann HG, Hornegger J, Wellein G. Pushing the limits for medical image reconstruction on recent standard multicore processors. *Int. J. High Perform. Comp. Appl.* 2012; doi:10.1177/1094342012442424. (Available online).
- [18] Intel architecture code analyzer, version 1.1.3. URL <http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/>.
- [19] McCalpin JD. Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter December 1995. URL [http://tab.computer.org/tcca/NEWS/DEC95/dec95\\_mccalpin.ps](http://tab.computer.org/tcca/NEWS/DEC95/dec95_mccalpin.ps).
- [20] McCalpin JD. STREAM: Sustainable memory bandwidth in high performance computers. *Technical Report*, University of Virginia, Charlottesville, VA 1991-2007. URL <http://www.cs.virginia.edu/stream/>, a continually updated technical report.
- [21] Hager G, Wellein G. *Introduction to High Performance Computing for Scientists and Engineers*. 1st edn., CRC Press, Inc.: Boca Raton, FL, USA, 2010.
- [22] Wolf-Gladrow D. *Lattice-Gas Cellular Automata and Lattice Boltzmann Models, Lecture Notes in Mathematics*, vol. 1725. Springer: Berlin, 2000.
- [23] Succi S. *The Lattice Boltzmann Equation – For Fluid Dynamics and Beyond*. Clarendon Press, 2001.
- [24] Chen S, Doolen G. Lattice Boltzmann Method for Fluid Flows. *Annu. Rev. Fluid Mech.* 1998; **30**:329–364.
- [25] Qian Y, d’Humières D, Lallemand P. Lattice BGK Models for Navier-Stokes Equation. *Europhys. Lett.* 1992; **17**(6):479–484.

- [26] Ziegler D. Boundary Conditions for Lattice Boltzmann Simulations. *J. Stat. Phys.* 1993; **71**(5/6):1171–1177.
- [27] Wellein G, Zeiser T, Hager G, Donath S. On the single processor performance of simple lattice Boltzmann kernels. *Computers & Fluids* 2006; **35**:910–919.
- [28] Zeiser T, Hager G, Wellein G. Benchmark analysis and application results for lattice Boltzmann simulations on NEC SX vector and Intel Nehalem systems. *Parallel Processing Letters* Aug 2009; **19**(4):491–511.